UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

Laboratorio di Reti

Anno 2018/2019

Turing: Relazione

Autore:
Monica Amico
516801
Corso B

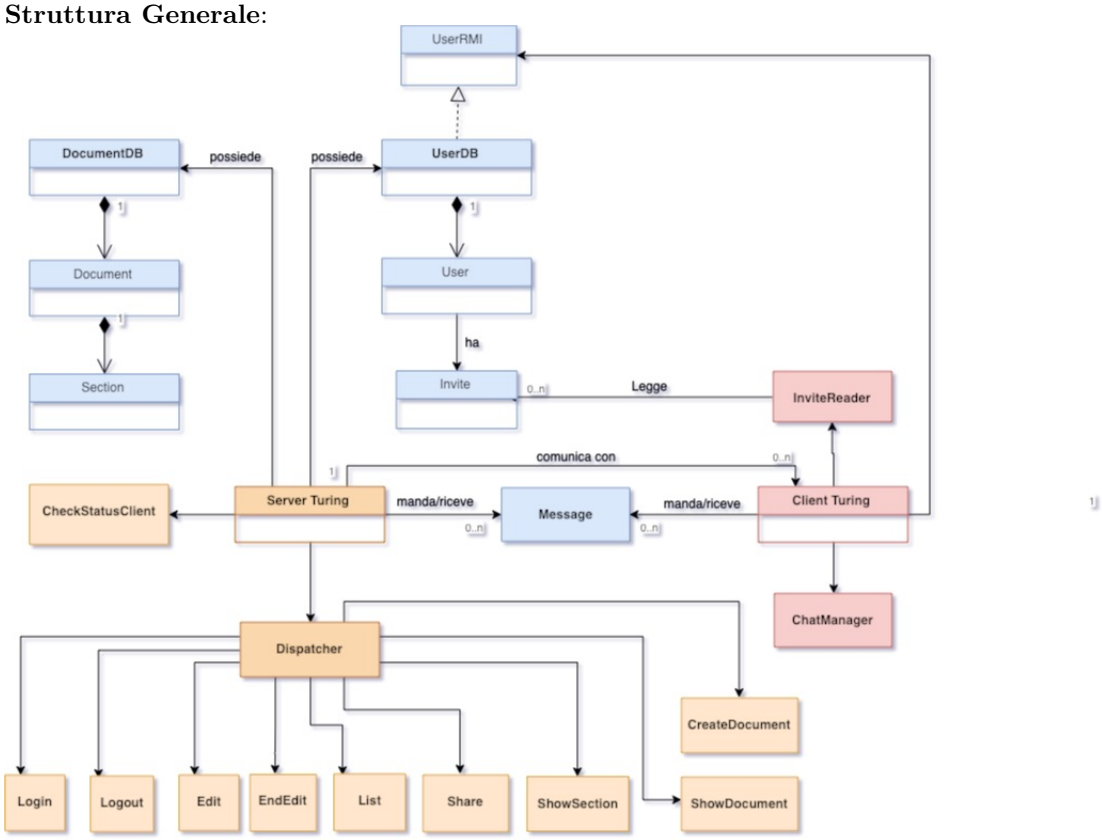
07 Febbraio 2019

Indice

1 Introduzione

Il progetto consiste nell'implementazione di *TURING*, uno strumento per l'editing collaborativo di documenti che offre un insieme di servizi minimale, tra cui: registrazione di utenti, editing di documenti, chat. Il progetto è stato sviluppato su MacOS, utilizzando l'ambiente di sviluppo Eclipse.

Struttura Generale:



2 Protocollo di comunicazione

I protocolli TCP, UDP e RMI sono stati utilizzati come da specifiche. Merita un particolare approfondimento lo scambio di messaggi tra client e server quando si utilizza il protocollo TCP. I messaggi possono essere astrattamente suddivisi in richieste dal client al server o risposte dal server al client.

2.1 La classe `Message`

Durante le *comunicazioni TCP*, client e server si scambiano oggetti della classe message. Un oggetto della classe message è composto da:

- `OPERATION op:` prende valori della enum `OPERATION` definita all'interno della classe. Rappresenta l'operazione richiesta dal client. Nei messaggi di richiesta indica l'operazione che il server deve effettuare. Nei messaggi di risposta indica a quale richiesta si sta dando risposta.
- `RESULT rs:` prende valori della enum `RESULT` definita all'interno della classe. Rappresenta l'esito di una operazione. Nei messaggi di richiesta è sempre null. Nei messaggi di risposta viene impostata con un valore indicante l'esito dell'operazione eseguita.
- `String username:` nome dell'utente che ha richiesto un operazione. Nei messaggi di richiesta questo campo ha sempre un valore diverso da null.
- `String password:` password dell'utente. Impostata soltanto nei messaggi di richiesta di tipo Login.
- `String destUsername:` questo campo ha valore diverso da null solo nei messaggi di richiesta per operazioni di "share", indica il nome dell'utente a cui il servere deve inoltrare l'invito.
- `String documentName:` campo che indica il nome del documento,è impostato solo nei messaggi di richiesta di tipo: creazione, share, show, edit, end-edit.
- `Set<Invite> listOfDocuments:` contiene una lista di oggetti della classe Invite. Utilizzato soltanto nei messaggi di risposta, nel caso in cui il client richiede una operazione di tipo "list".
- `int numOfSection:` campo che indica il numero di sezione. Impostato solo nei messaggi di richiesta di tipo: creazione documento, show (-1 se si desidera visualizzare l'intero documento), edit, end-edit.

La classe contiene metodi per serializzare, deserializzare, inviare e ricevere oggetti di questo tipo:

- `boolean sendMessage(SocketChannel client, Message message);`
- `Message receiveMessage(SocketChannel client).`

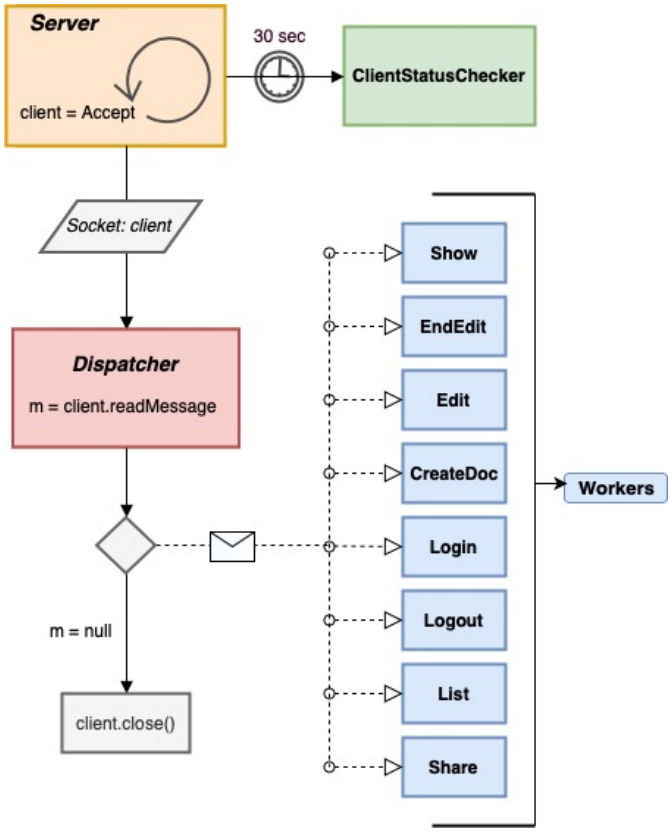
3 Il Server

3.1 Struttura e flusso di lavoro

Il server progettato è multithreaded ed effettua il multiplexing dei canali mediante NIO (in lettura). E'organizzato in varie componenti principali:

- Server: `Turing;`
- Un `Dispatcher;`
- Un Thread-Timer: `CheckClientStatus;`
- `Workers` (uno per operazione).

Al suo avvio il Server Turing, inizializza le principali strutture dati e avvia i thread: Dispatcher e CheckClientStatus. Mentre per i Workers, sarà il thread Dispatcher che si occuperà dell'avvio del worker adatto, in base all'operazione richiesta dal client.



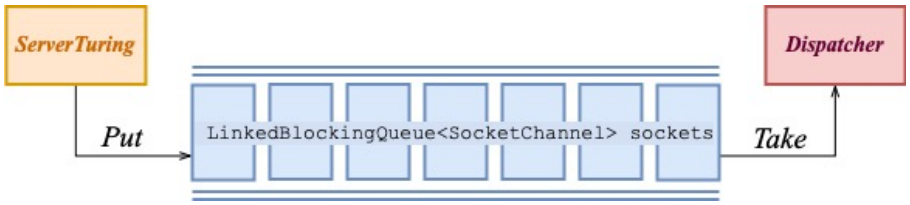
3.2 Strutture dati

- **DocumentDB:** Database contenente tutti i documenti presenti nel server, è rappresentato da una `ConcurrentSkipListSet<Document>`.
  Questa implementazione fornisce il costo medio pari a log(n) per operazioni come: contains, add, remove e loro varianti. Inoltre le operazioni vengono eseguite in modo sicuro simultaneamente da più thread. Gli iteratori non generano ConcurrentModificationException, ovvero possono procedere in concomitanza con altre operazioni.
- **Document:** struttura che rappresenta un singolo documento. Un documento è composto da uno o più file chiamati sezioni. (`Section`).
  Ogni documento contiene:
  1. `String docName`: nome del documento;
  2. `String owner`: nome del creatore del documento;
  3. `ConcurrentSkipListSet<Section> sections`: sezioni del documento.
  4. `ConcurrentSkipListSet<String> autorizedUsers`: lista di nomi degli utenti autorizzati;
  5. `AtomicInteger group_port`: porta della chat di gruppo.
  6. `AtomicInteger numEditing`: numero di utenti che in un certo istante stanno modificando il documento.
- **Section:** rappresenta una sezione di un documento. contiene:
  1. `String fileName`: nome del file sul disco
  2. `String activeUser`: nome dell'utente che sta modificando la sezione, null se nessun utente la sta modificando.
  3. `int numSec`: numero della sezione.
- **UserRMI:** interfaccia utilizzata per accedere al metodi remoti.
- **UserDB:** Database contenente tutti gli utenti registrati. E' rappresentato da una `ConcurrentSkipListSet<User>`. Implementa l'interfaccia UserRMI.
- **User:** struttura che contiene i dati di un singolo utente. E' composto da:
  1. `String username`;
  2. `String password`;
  3. `SocketChannel onLine`;
  4. `ConcurrentSkipListSet<Invite> documents`: lista di documenti che può modificare.
  5. `ConcurrentSkipListSet<Invite> invites`: lista di inviti ricevuti, viene riempita nel caso in cui l'utente riceve degli inviti mentre è offline.
- **Invite:** contiene nomi del documento e del creatore.

3.3 Server: Turing.java

Il Server implementa il NIO utilizzando un selettore. Inizialmente nel selettore viene registrata una socket utilizzata per l'accettazione di una connessione da parte di un client. Successivamente all'accettazione di una connessione, la socket generata viene registrata nel selettore per operazioni di lettura. Quando una di queste socket risulta pronta, viene inserita in una coda di SocketChannel: `sockets` condivisa con il dispatcher, e cancellata dal selettore.

Il Server quindi ha il compito di accettare nuove connessioni e permettere al Dispatcher di prelevare una socket per leggere il messaggio contenuto in essa.



3.4 Dispatcher

Il Dispatcher ha il compito di prelevare le socket dalla coda, leggere il messaggio ricevuto (oggetto della classe Message) e in base al campo operazione di esso, smistarlo al worker adatto. In caso di errore la socket prelevata viene chiusa.

3.5 Workers

Esiste un worker per ogni possibile operazione, tale scelta è stata fatta per avere una struttura modulare, facilitando l'individuazione degli errori e ottenendo un grado elevato di disaccoppiamento (low coupling). Ogni worker controlla vari campi del messaggio.
Nella richiesta di tipo Login, viene creata una connessione TCP, con porta: `(Turing.LOGIN_PORT)`, che resterà aperta per tutto il ciclo di vita del client. Tale socket sarà utilizzata per notificare la ricezione di inviti di collaborazione da parte di altri utenti. Mentre per ogni altra richiesta, viene aperta e chiusa al termine dell'operazione, una connessione TCP, con porta: `(Turing.REQUEST_PORT)`.

- **Login:** controlla se esiste un utente con l'username ricevuto e se la password ricevuta è corretta.
  Se non esiste l'utente o la password non è corretta, viene inviato un messaggio con esito negativo al client e la connessione viene chiusa, altrimenti imposta la variablie `onLine` dell'utente uguale alla socket corrente, in modo tale da mantenere una corrispondenza tra utente e socket, e invia un messaggio con esito positivo al client. Tale connessione resterà aperta fino a quando il client non richiederà esplicitamente il logout/exit o fino a quando il thread CheckClientStatus non la chiuderà (caso in cui il client è chrashato). Da questo momento il client non scriverà più su questa socket,

ma la utilizzerà solamente in lettura, solamente il Server potrà utilizzarla in scrittura per mandare gli inviti destinati al client.

- **Logout:** Se l'utente è online, imposta la variabile `onLine a null` e chiude la connessione permanente associata all'utente. Successivamente invia l'esito alla socket della richiesta e la chiude.

- **CreateDocument:** Se l'utente esiste, e il documento non esiste già, viene creato il documento con N sezioni e viene inserito nel DocumentDB e nella lista documenti dell'utente. Successivamente invia l'esito (true/false) a chi ha richiesto l'operazione e chiude la socket della richiesta. Durante la creazione non viene creato nessun file sul disco del server, verranno creati quando saranno necessari, cioè al momento di una modifica/visualizzazione.

- **Edit:** effettua i relativi controlli:

  1. controlla se l'utente esiste;
  2. controlla se il documento esiste;
  3. controlla che l'utente abbia i diritti necessari per modificare il documento;
  4. controlla che il numero della sezione da modificare sia corretto.

  A questo punto:

  - Uno dei controlli fallisce: invia l'esito negativo al client e chiude la connessione.
  - I controlli sono tutti andati a buon fine:
    chiama la funzione `startEdit` sulla sezione desiderata, che se possibile inserirà il nome dell'utente nella variabile activeUser e crea il file (se non esiste già). Se la startEdit fallisce invia un esito negativo al client e chiude la connessione; altrimenti:
    1. invia l'esito positivo;
    2. legge il contenuto del file e lo invia;
    3. invia il numero di porta relativo alla chat del documento.
    4. chiude la connessione.

- **EndEdit:** esegue i regolari controlli, e inoltre controlla se l'utente che ha richiesto l'operazione sta modificando la sezione del documento che desidera salvare. Se i controlli vanno a buon fine, questo thread legge il contenuto del file inviato dal client e sovrascrive il vecchio file con il nuovo contenuto, successivamente invia l'esito positivo al client. Altrimenti invia un esito negativo. Al termine dell'operazione chiude la connessione TCP.

- **List:** controlla se l'utente che ha richiesto la lista è registrato e invia la lista di documenti. Al termine chiude la connessione.

- **ShowDocument:** controlla se l'utente e il documento esistono e se l'utente ha i diritti necessari, in caso di esito positivo crea il documento unendo le varie sezioni. Preleva il contenuto di ogni sezione e successivamente lo inserisce nel file che rappresenta il documento. Manda l'esito al client e il contenuto del documento richiesto.
- **ShowSection:** controlla se l'utente, il documento e la sezione esistono, e se l'utente ha i diritti necessari. Invia l'esito al client e se questo è positivo, invia il contenuto della sezione tramite un altro messaggio.
- **Share:** operazione per inviare un'invito a un'utente. Controlla se l'utente invitato e il documento esistono, e se l'utente che ha richiesto l'operazione esiste ed è il creatore del documento. Se i controlli vanno a buon fine, crea l'invito e controlla lo stato dell'utente destinatario:
  - Offline: l'invito viene inserito nella lista degli inviti pendenti del destinatario.
  - Online: l'invito viene inviato sulla socket permanente del destinatario (Socket del Login).

  Al termine invia l'esito a chi ha richiesto l'operazione.

Non è presente un worker per la registrazione perchè viene effettuata dal client con RMI.

3.6 Thread-Timer: CheckStatusClient

Questo task temporale viene attivato ogni trenta secondi e ha il compito di disconnettere gli utenti i cui client sono stati chiusi senza effettuare un esplicita operazione di logout/exit o che sono terminati in maniera imprevista. Per riconoscere gli utenti su cui operare questo thread effettua una read sulla socket degli inviti, se la read restituisce -1 l'utente viene disconnesso in maniera sicura, terminando eventuali operazioni di modifica in sospeso (lasciando inalterato il documento).

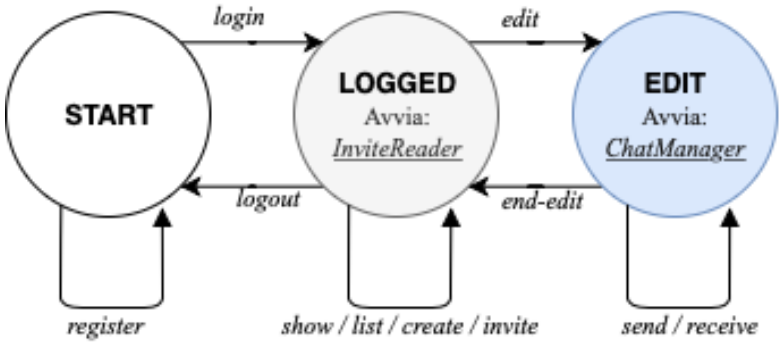
3.7 Gestione della concorrenza

Per la gestione della concorrenza sono state utilizzate strutture adatte, come: code bloccanti (LinkedBlockingQueue), liste concorrenti (ConcurrentSkipListSet), e metodi synchronized. Non è stato fatto utilizzo di lock esplicite.

4 Il Client

4.1 Struttura e flusso di lavoro

Il client possiede una componente principale, `ClientTuring.java` e due componenti ausiliarie, attivate al momento opportuno:
`InviteReader.java` e `ChatManager.java`.
Al suo avvio il client inizializza le sue strutture dati e successivamente esegue un ciclo dove ad ogni iterazione richiede un comando. Se il comando digitato è uguale a `exit` il client termina.



4.2 Gesione della chat

La gestione della chat, tra utenti che modificano uno stesso documento, viene effettuata quasi interamente dal client. L'unico compito del server è generare un numero di porta, e assicurarsi che tutti i partecipanti, a una stessa chat, ricevano lo stesso numero di porta. Quando un client riceve un esito positivo dopo aver richiesto l'operazione di edit, crea la socket della chat e si aggiunge al gruppo multicast. La chat verrà chiusa dal client stesso, dopo una successiva end-edit. Ovviamente questa soluzione funziona finchè i client sono sulla stessa macchina, una soluzione alternativa sarebbe quella di creare la socket nel server e inviare l'indirizzo e la porta al client.

4.3 Utilizzo

Registrazione: `turing register <nome> <password>`
Login: `turing login <nome> <password>`
Logout: `turing logout`
Create Document: `turing create <doc> <nsez>`
Edit: `turing edit <doc> <nsez>`
End-Edit: `turing end-edit`
List: `turing list`
Show Document: `turing show <doc>`
Show Section: `turing show <doc> <nsez>`
Share: `turing share <doc> <user>`

5 Note sulla suddivisione dei file

Files del server:

- Turing.java
- Dispatcher.java
- CheckClientStatus.java
- Login.java
- Logout.java
- Edit.java
- EndEdit.java
- ShowDocument.java
- ShowSection.java
- Share.java
- List.java
- CreateDocument.java
- User.java
- UserDB.java
- Document.java
- DocumentDB.java
- Section.java
- UserRMI.java
- Invite.java
- Message.java

Files del client:

- ClientTuring.java
- InviteReader.java
- ChatManager.java
- Invite.java
- Message.java

6 Directories e file

Per ogni sezione viene memorizzato un file, con nome:
`nomeDocumento_numeroDezione`, (es: doc_0). Le sezioni sono numerate da 0 a *Nsezioni -1*. Tutti i files creati dal Server saranno contenuti nella cartella:
`TURING/ServerDocuments/`.
I files dei client sono suddivisi per utente, si possono trovare nella cartella:
`TURING/ClientDocuments/nome_utente/`